시스템 프로그래밍

Assignment2-2

Class : A

Professor : 김태석 교수님

Student ID : 2022202065

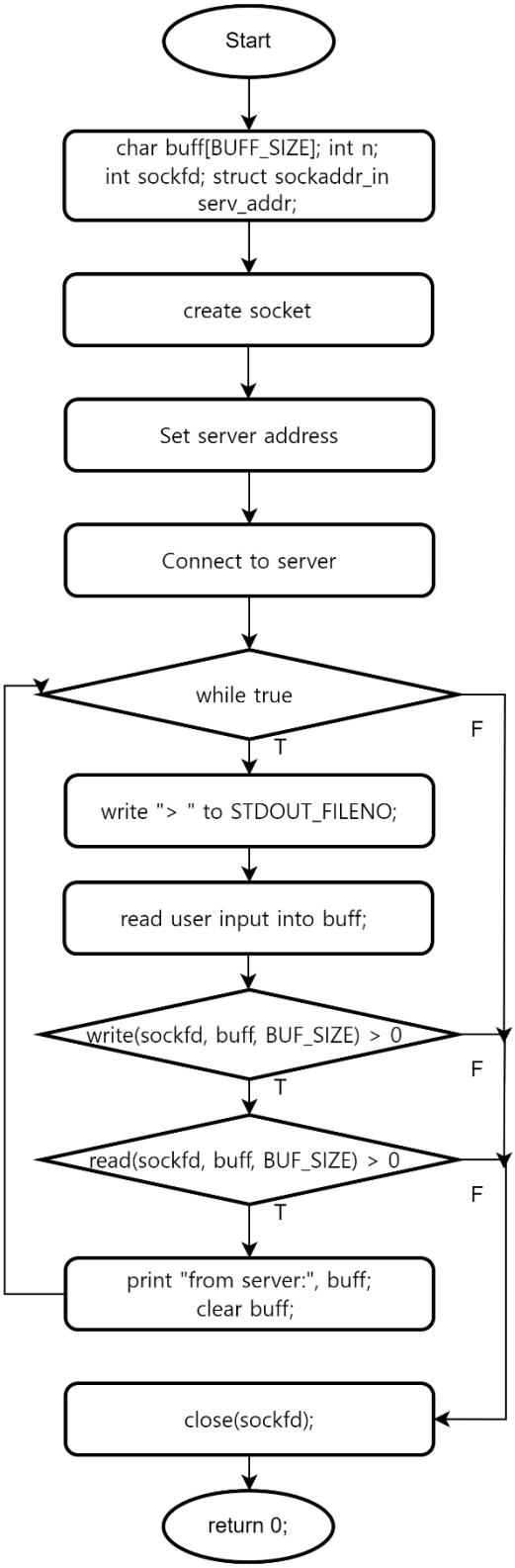
Name : 박나림

Introduction

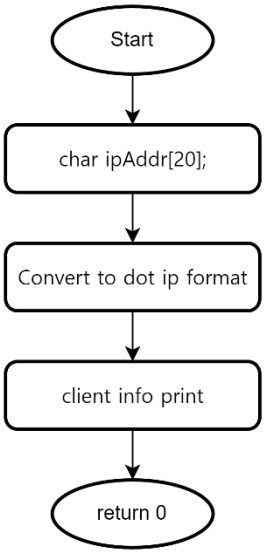
FTP 를 구현하는 프로젝트 중 fork 함수 및 시그널 관련 함수를 다루는 과제이다. 클라이언트와 연결되면, 서버에서는 새로운 프로세스를 생성하여 각각 작업을 진행한다. 부모 프로세스에서는 연결된 클라이언트의 ip, port 정보를 출력하고, 자식 프로세스에서는 클라이언트로부터 받은 문자열을 다시 전송하는 과정을 반복한다. 이때 QUIT 이 들어오면 해당 클라이언트와의 연결을 종료하며, SIGALRM 시그널을 호출하여 해당 프로세스도 종료되는 것이다. 또한 이러한 프로세스들이 새로 생성될 때마다 각각의 PID 를 출력하도록 한다. 따라서 이번 과제를 통해 클라이언트와 서버간의 연결 과정에서 fork, 시그널 함수를 다루는 것이 목표이다.

Flow chart

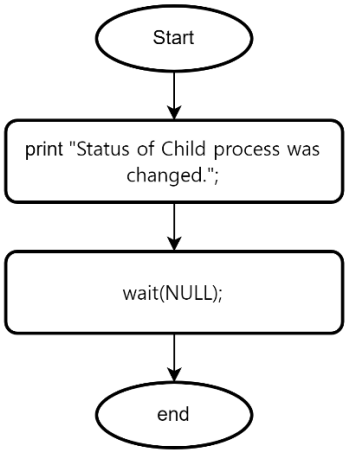
cli.c – main()



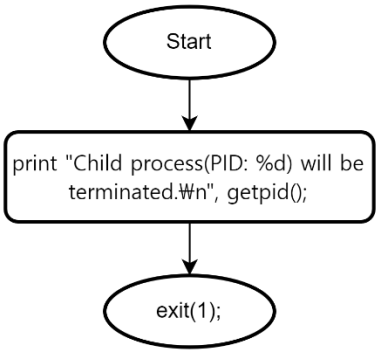
srv.c – client_info



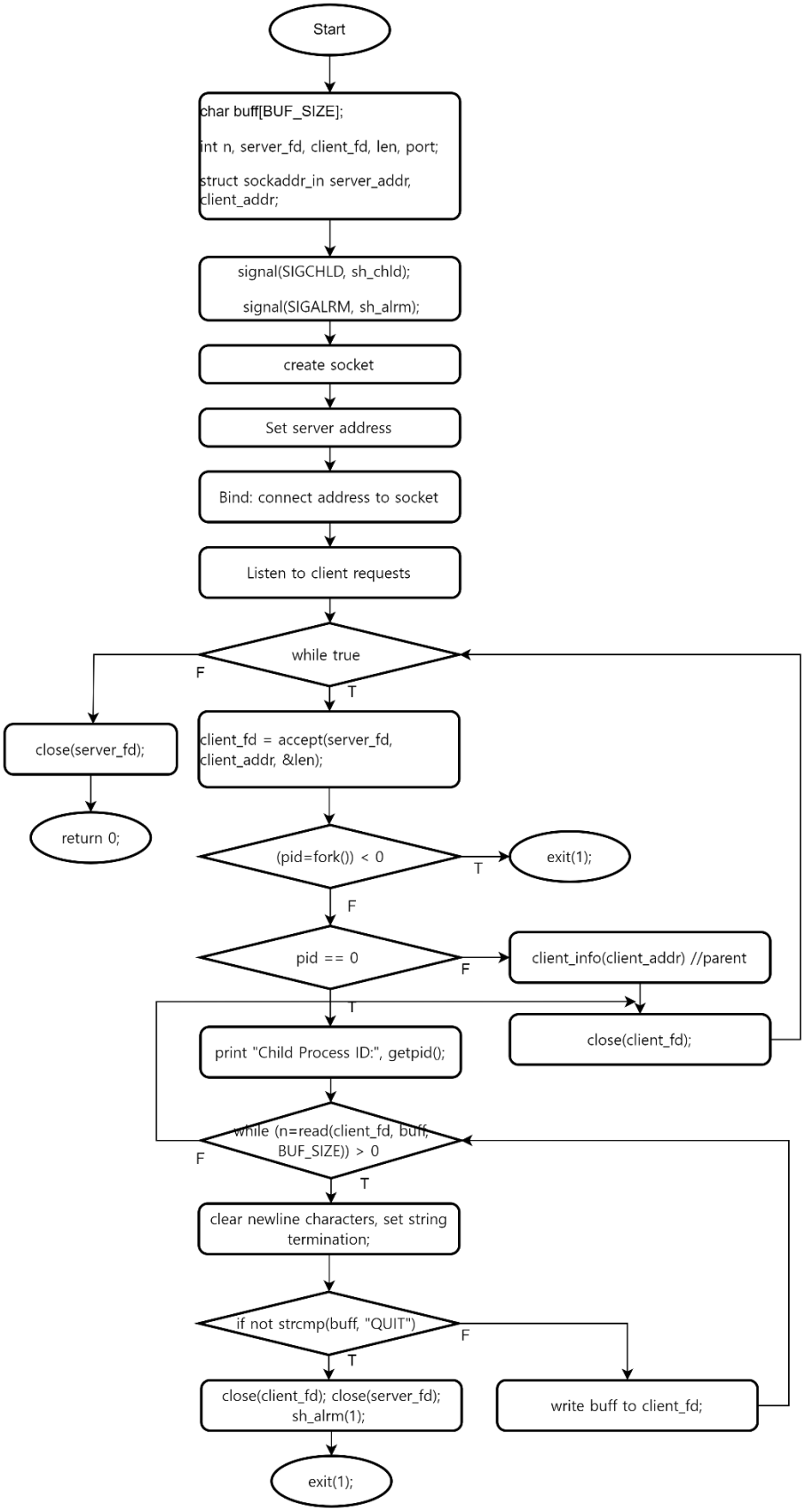
srv.c – sh_chld()



srv.c – sh_alrm()



srv.c – main()


Start
char buff[BUF_SIZE];
int n, server_fd, client_fd, len, port;
struct sockaddr_in server_addr, client_addr;
signal(SIGCHLD, sh_chld);
signal(SIGALRM, sh_alrm);
create socket
Set server address
Bind: connect address to socket
Listen to client requests
while true
F
T
close(server_fd);
return 0;
client_fd = accept(server_fd, client_addr, &len);
(pid=fork()) < 0
T
exit(1);
F
pid == 0
F
client_info(client_addr) //parent
T
close(client_fd);
print "Child Process ID:", getpid();
while (n=read(client_fd, buff, BUF_SIZE)) > 0
F
T
clear newline characters, set string termination;
if not strcmp(buff, "QUIT")
F
T
close(client_fd); close(server_fd); sh_alrm(1);
write buff to client_fd;
exit(1);

Pseudo code

cli.c

```
int main(integer argc, array argv) {
    char buff[BUF_SIZE];
    int n;
    int sockfd;
    struct sockaddr_in serv_addr;
    sockfd = socket(AF_INET, SOCK_STREAM, 0) // create socket
        // Set server address
    set serv_addr to 0;
    serv_addr.sin_family = AF_INET; // set family
    serv_addr.sin_addr.s_addr = inet_addr(argv[1]); // set IP address
    serv_addr.sin_port = htons(atoi(argv[2])); // set port
        // Connect to server
    connect(sockfd, serv_addr, sizeof(serv_addr));
        // Read user input and send to server, receive results and print
    while true {
        write "> " to STDOUT_FILENO;
        read user input into buff; // read user input
        if write(sockfd, buff, BUF_SIZE) > 0 then          // send string to server
            if read(sockfd, buff, BUF_SIZE) > 0 then       // receive and print string again
                print "from server:", buff;
                clear buff;              // buffer clear
            else   // quit if string is not received
                break;
            end if
        else         // quit if the string cannot be sent to the server
```

```
                break;
            end if
        } //end while
        close(sockfd);
        return 0;
}
```

srv.c

```
int client_info(pointer client_addr) {
        char ipAddr[20]; // ip address
        Convert to dot ip format;
        print Client info;
        return 0;
}
```

```
void sh_chld(integer signum) {
        print "Status of Child process was changed.";
        wait(NULL); // Parent process waits for child process to terminate;
}
```

```
void sh_alrm(int signum) {
            print "Child process(PID: %d) will be terminated.₩n", getpid();
            exit(1);
}
```

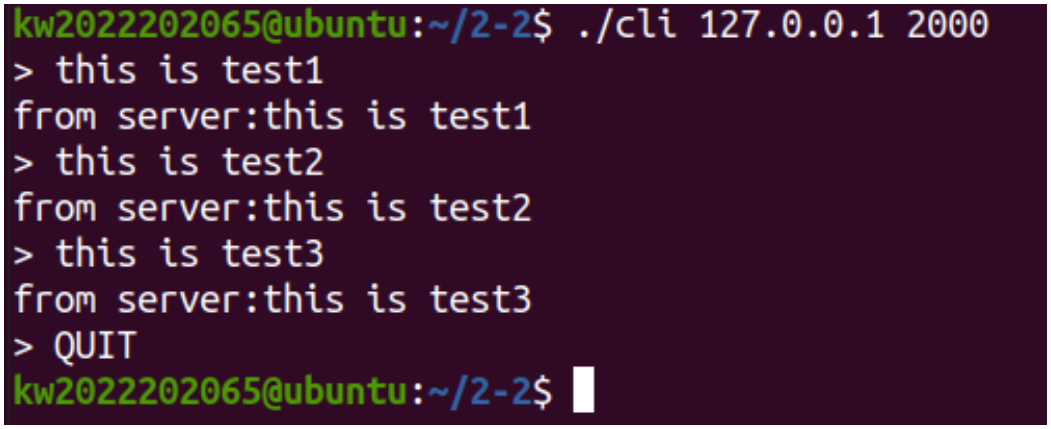
```
int main(integer argc, array argv) {
        char buff[BUF_SIZE];
        int n, server_fd, client_fd, len, port;
        struct sockaddr_in server_addr, client_addr;
        signal(SIGCHLD, sh_chld);              // applying signal handler(sh_alrm) for SIGALRM
```

```
signal(SIGALRM, sh_alrm);              // applying signal handler(sh_chld) for SIGCHLD
server_fd = socket(PF_INET, SOCK_STREAM, 0);          // create socket
      // Set server address
set server_addr to 0;
server_addr.sin_family = AF_INET;                          // set family
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);     // set IP address
server_addr.sin_port = htons(atoi(argv[1]));                // set port
      // Bind: connect address to socket
bind(server_fd, server_addr, sizeof(server_addr));
      // Listen to client requests
listen(server_fd, 5);
      // Connect to client, read and execute
while true {
     int pid;     // process ID
     len = sizeof(client_addr);
     client_fd = accept(server_fd, client_addr, &len);
      // Fork
     if (pid=fork()) < 0 then          // fork error
          print "Error: fork() error!!";
          exit(1);
     else if pid == 0 then             // child process
          print "Child Process ID:", getpid();
          while (n=read(client_fd, buff, BUF_SIZE)) > 0 { // read client's string
               clear newline characters, set string termination;
               if not strcmp(buff, "QUIT") then             // program quit
                    close(client_fd);
                    close(server_fd);
                    sh_alrm(1);         // 1 second
```

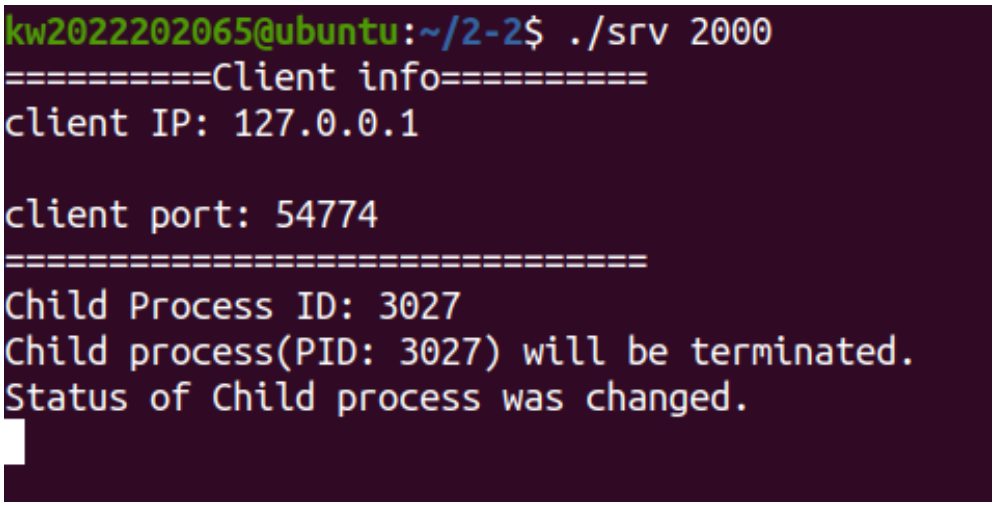
```
                end if
                write buff to client_fd;            // send to client
            } //end while
        else          // parent process
            if client_info(client_addr) < 0 then // display client ip and port
                print "Error: client_info error!!";
            end if
        end if
        close(client_fd);
    } //end while
    close(server_fd);
    return 0;
}
```

결과화면

1) 클라이언트에서 서버와 연결한 뒤, 차례대로 test 문구를 보내면 바로 서버에서 다시 전송해서 출력한다. 이때 서버는 새로운 프로세스를 생성해서 문자열 반환 작업을 하게 되는데, 마지막처럼 QUIT 을 입력하면 해당 클라이언트와 연결이 종료된다. 따라서 클라이언트의 결과화면은 아래와 같다.



서버에서는 연결직후 parent 프로세스에서 클라이언트의 ip, port 정보를 출력한다. 이후 프로세스가 생성될 때마다 child process ID 를 같이 출력한다. 클라이언트에서 QUIT 이 들어오면 해당 child process 는 종료된다. 이때 SIGALRM 을 호출하기 때문에 해당 프로세스가 종료되었다는 메시지가 출력된다.



2) 위와 같은 상황에서, 서버 프로그램은 종료되지 않는다. 따라서 클라이언트를 다시 연결하면 그대로 다시 진행할 수 있게 된다. 아래와 같이 첫번째 test 를 입력한 뒤 QUIT 을 하여 서버에서 해당 프로세스가 종료되었다는 메시지가 출력되면, 클라이언트는 종료된다. 그리고 다시 클라이언트에서 연결을 요청하고 test 를 입력하면 서버에서는 이어서 새 프로세스를 만들어서 다시 작업을 진행할 수 있게 된다. child process ID 를 통해 이어서 만들어진 새 프로세스임을 확인할 수 있다.

```
kw2022202065@ubuntu:~/2-2$ ./cli 127.0.0.1 2000
> this is test1
from server:this is test1
> QUIT
kw2022202065@ubuntu:~/2-2$ ./cli 127.0.0.1 2000
> this is test2
from server:this is test2
> QUIT
kw2022202065@ubuntu:~/2-2$
```

```
kw2022202065@ubuntu:~/2-2$ ./srv 2000
==========Client info==========
client IP: 127.0.0.1

client port: 60256
================================
Child Process ID: 1968
Child process(PID: 1968) will be terminated.
Status of Child process was changed.
==========Client info==========
client IP: 127.0.0.1

client port: 57430
================================
Child Process ID: 1970
Child process(PID: 1970) will be terminated.
Status of Child process was changed.
```

고찰

클라이언트와 서버를 구현하고 나서, 실행을 시켜 보니 문자열이 제대로 출력되지 않았다. 처음엔 클라이언트에서 문자열을 입력 받고 서버로 보내는 과정에서 문자열 끝을 설정을 안 해서 그런 것이라고 생각하였다. 하지만 그 부분을 수정하여도 여전히 버퍼에 다른 문자가 섞인 듯이 출력되었다. 그래서 클라이언트가 아니라 서버 프로그램 내에서 문자열 끝 설정 코드를 추가하였다. read 를 통해 클라이언트에서 문자열을 받으면, 개행문자가 있는지 확인하고 있을 시에 해당 문자를 '/0'으로 바꾸었다. 그렇게 하니 다른 문자는 없어졌지만, 이번엔 출력이 바로 안되었다.

입력을 받으면 그때마다 서버에서 돌아오는 문자열을 출력해야 하는데, 계속 입력만 받다가 QUIT 이 들어오고 나서야 종료될 때 한번에 from server 문자열들이 출력되었다. 그래서 while 문 같은 반복문 내에서 문제가 있는 것이라 생각하였는데, 수정해도 달라지지 않았다. 이에 하나씩 코드를 살펴보다가 write 에서 길이를 설정하는 부분을 수정하였다. 원래 BUF_SIZE 또는 sizeof(buff)로 하였었는데, strlen(buff)로 하니 해당 오류를 덜 발생하게 수정할 수 있었다. 버퍼의 크기 자체가 아니라, '/0'전까지의 문자 길이를 보내는 것이 확실하기 때문인 것 같다. 자세한 버퍼관리 방법은 더 공부해볼 것이다.

또한 이번 과제를 통해 fork 함수 및 시그널 관련 함수 사용법에 대해 더 잘 이해할 수 있었다. fork 를 통해서 부모와 자식 프로세스가 각각 다른 일을 수행할 수 있는 부분에서는 단순한 조건문과 비슷해보이지만, 클라이언트와 자식 프로세스간의 연결이 끊겨도 부모 프로세스는 여전히 진행되면서 새로 자식 프로세스를 생성하여 작업할 수 있다는 점이 새로웠다. 시그널 함수에서는 처음에 잘 이해를 못하다가, 다시 공부해보면서 방법을 익힐 수 있었다. 먼저 함수를 등록하는 과정을 거쳐야 사용할 수 있다는 점도 깨달았다.

Reference

강의자료 참고